

A GPU-Based Solution to Fast Calculation of Betweenness Centrality on Large Weighted Networks

Rui Fan¹, Ke Xu¹ and Jichang Zhao^{2,*}

¹State Key Lab of Software Development Environment, Beihang University

²School of Economics and Management, Beihang University

*Corresponding author: jichang@buaa.edu.cn

Abstract

Recent decades have witnessed the tremendous development of network science, which indeed brings a new and insightful language to model real systems of different domains. Betweenness, a widely employed centrality in network science, is a decent proxy in investigating network loads and rankings. However, the extremely high computational cost greatly prevents its applying on large networks. Though several parallel algorithms have been presented to reduce its calculation cost on unweighted networks, a fast solution for weighted networks, which are in fact more ubiquitous than unweighted ones in reality, is still missing. In this study, we develop an efficient parallel GPU-based approach to boost the calculation of betweenness centrality on very large and weighted networks. Comprehensive and systematic evaluations on both synthetic and real-world networks demonstrate that our solution can arrive the performance of 30x to 150x speedup over the CPU implementation by integrating the work-efficient and warp-centric strategies. Our algorithm is completely open-sourced and free to the community and it is public available through <https://dx.doi.org/10.6084/m9.figshare.4542405>. Considering the pervasive deployment and declining price of GPU on personal computers and servers, our solution will indeed offer unprecedented opportunities for exploring the betweenness related problems in network science.

1 Introduction

Recent years, the network science, a multidiscipline research area, is concentrated by researchers from different backgrounds such as computer science, biology and physics. In these studies, betweenness centrality (BC) is always applied as a critical metric to measure nodes or edges' significance [1, 2, 3]. For example, Girvan and Newman developed a community detection

algorithm based on edge betweenness centrality [4], Leydesdorff applied centrality as an indicator of the interdisciplinarity of scientific journals [5] and Motter and Lai established a model of cascading failures with node load being its betweenness [6]. However, the extremely high time and space complexity of calculating betweenness centrality greatly limits its applying on large networks. Before the landmark work of Brandes [7], the algorithm for computing betweenness centrality requires $O(n^3)$ time and $O(n^2)$ space. While Brandes reduced the complexity to $O(n+m)$ on space and $O(nm)$ and $O(nm + n^2 \log(n))$ on time for unweighted and weighted networks, respectively, where n is the number of vertices and m is the number of edges [7]. However, this improved algorithm still can not satisfy scientific computation requirements in the present information explosion era as more and more unexpected large networks emerge, such as online social networks, gene networks and collaboration networks. For example, Twitter possesses hundreds of millions active users which construct a huge online social network. However, a weighted network with one million nodes may take about one year to calculate its betweenness centrality using Brandes' algorithm, which is an unbearable cost. Because of this, there is a pressing need to develop faster BC algorithm for explorations of diverse domains.

GPU general computing, which provides excellent parallelization, achieves higher performance compared to traditional CPU sequential algorithms in many issues including network science [8, 9, 10, 11]. CUDA is the most popular GPU-computing framework developed by NVIDIA corporation and some researchers have even parallelized Brandes's algorithm by using it [12, 13, 14]. However, previous works concentrated on unweighted networks for simplification, but to our best knowledge, most realistic networks are weighted ones. The most significant difference of BC algorithm on unweighted and weighted networks is the shortest path segment. In weighted networks, Dijkstra algorithm should be used to solve the single source shortest path (SSSP) problem rather than Depth First Search (DFS) algorithm. Many efforts in previous work have been devoted to the GPU version of SSSP problem using the well-known Dijkstra algorithm [15, 16, 17, 18]. Although these algorithms have been presented and developed, establishing a parallel version of betweenness centrality algorithm on weighted networks is nontrivial because the original SSSP algorithm have to be modified in many critical points for this task and to our best knowledge, a proper and fast solution is still missing. Aiming at filling this vital gap, we propose a fast solution using CUDA to calculate BC on large weighted networks based on previous GPU BC algorithms and SSSP algorithms in this paper.

To make our algorithm more efficient, we make efforts to optimize it by employing several novel techniques to conquer the influence of irregular network structures. Real-world networks have many characters which could deteriorate the performance of GPU parallelization algo-

rithms. For example, the frontier set of nodes is always small compared to the total number of vertices, especially for networks with great diameters. In the meantime, the majority of nodes do not need to be inspected in each step, hence processing all vertices simultaneously in traditional algorithms is wasteful. McLaughlin and Bader proposed a work-efficient strategy to overcome this problem [14]. Another well-known issue is that the power-law degree distribution in realistic networks brings in serious load-imbalance. Several methods were proposed in previous study to conquer this problem, e.g., Merrill et al. employed edge parallel strategy to avoid load-imbalance [8] and Hong et al. dealt with this problem by using warp technique [19]. In this paper, we systematically investigate the advantages and drawbacks of these previous methods and implement them in our algorithm to solve the above two problems. Experiments on both real-world and synthetic networks demonstrate that our algorithm outperforms the baseline GPU algorithm significantly. Our main contributions are listed as follows:

- Based on previous GPU parallel SSSP and betweenness centrality algorithms, we propose an efficient algorithm to calculate betweenness centrality on weighted networks, which achieves $30\times$ to $150\times$ speedup over the best existing CPU algorithm on realistic networks.
- We compare the traditional node-parallel method to the work-efficient version and the warp-centric method. Experiments on realistic networks and synthetic networks demonstrate that the combination of the two strategies works better than others, which achieves $2.65\times$ speedup over the baseline method on realistic networks.
- We package our algorithm to a useful tool which can be used to calculate both node and edge betweenness centrality on weighted networks. Researchers could apply this tool to conveniently calculate BC on weighted networks fast, especially on large networks. The source code is publicly available through <https://dx.doi.org/10.6084/m9.figshare.4542405>.

2 Background

First we briefly introduce the well-know Brandes’s algorithm and Dijkstra algorithm based on the preliminary definitions of network and betweenness centrality.

2.1 Brandes’s algorithm

A graph can be defined as $G(V, E)$, where V is the set of vertices, and E is the set of edges. An edge is a node pair (u, v, w) , which means that there is a link connecting nodes u and v , and its

weight is w . If the edge (u, v) exists, it can be traversed from u to v and from v to u because we only focus on undirected graphs in this paper. However, our algorithm can be expanded to directed graph version easily. A path $P = (s, \dots, t)$ is defined as a sequence of vertices connected by edges, where s is the starting node and t is the end node. The length of P is the sum of the weights of the edges involved in P . $d(s, t)$ is the distance between s and t , which represents the minimum length of all paths connecting s and t . σ_{st} denotes the number of shortest paths from s to t . According to the definition, we have $d(s, s) = 0$, $\sigma_{ss} = 1$, $d(s, t) = d(t, s)$ and $\sigma_{st} = \sigma_{ts}$ for undirected graph. $\sigma_{st}(v)$ denotes the number of shortest paths from s to t where v lies on. Based on these definitions, the betweenness centrality can be defined as

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}. \quad (1)$$

From the above definitions, the calculation of betweenness centrality can be naturally separated into the following two steps:

1. Compute $d(s, t)$ and σ_{st} for all node pairs (s, t) ,
2. Sum all pair-dependencies,

in which pair-dependency is defined as $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. The first step consumes $O(mn)$ and $O(mn + n^2 \log(n))$ time for unweighted and weighted graph respectively, therefore the bottleneck of this algorithm is the second step, which requires $O(n^3)$ time. Brandes's developed a more efficient BC algorithm which requires $O(mn)$ time for unweighted graph, and $O(mn + n^2 \log(n))$ time for weighted graph. The critical point is that the dependency of a node v when the source node is s is $\delta_s(v) = \sum_{u: v \in P_s(u)} \frac{\sigma_{sv}}{\sigma_{su}} (1 + \delta_s(u))$. Applying this equation, we can accumulate the dependencies after computing the distance and number of shortest paths from a source vertex s to all other vertices, rather than after computing all pair shortest paths.

We can develop a parallel version based on Brandes's algorithm for unweighted graph because the graph is always traversed as a tree by using DFS algorithm. Given a source node s , the root of the tree is s and the tree produced by DFS method in the first step. In the second step, dependencies related to source node s are calculated from the bottom to the root of the tree and the nodes at the same level are isolated and have no influence to each other. As a result, the parallel version can explore nodes at the same level simultaneously in both of the two steps, which will essentially boost the calculation.

2.2 Dijkstra algorithm

Dijkstra algorithm [20] and Floyd-Warshall algorithm [21] are commonly employed to solve shortest path problems. While Dijkstra algorithm is more adaptable to betweenness centrality problem because Brandes’s algorithm accumulates dependencies after computing single source shortest paths (SSSP), rather than finding and storing all pair shortest paths. Dijkstra algorithm applies greedy strategy to solve SSSP. In this algorithm, the source node is s and if the shortest path from s and another node u is achieved, u will be settled. According to be settled or not, all nodes in graph G could be separated into two sets, which are settled vertices S and unsettled vertices U . An array D is used to store tentative distances from s to all nodes. At first, $D(s) = 0$ and $D(u) = \infty$ for all other nodes. And the source node s is settled and considered as the frontier node to be explored. In the second step, for every node $u \in U$ and the adjacent frontier node f , if $D[f] + w(f, u) < D[u]$, $D[u]$ will be updated to $D[f] + w(f, u)$. Then the node $v \in U$ that has the smallest distance value will be settled and considered as the new frontier node and then the procedure goes back to the second step. The algorithm finishes when all nodes are settled. From the above description, Dijkstra algorithm has no parallel character as it picks one frontier node in each iteration. But this restriction can be loosed that several frontier vertices can be explored simultaneously which is similar to DFS parallel approach.

3 GPU-based Algorithm

3.1 Parallel betweenness centrality algorithm

In this section, we introduce the details of our GPU version BC algorithm on weighed graph. Firstly, we apply *Compressed Sparse Row* (CSR) format, which is widely used in graph algorithms, to store the input graph [22, 18]. It is space efficient that both of the vertex and edge consume one entry, and it is convenient to perform the traversal task on GPU. Moreover, edges related to the same vertex store consecutively in memory which makes warp-centric technique more efficient. For storing weighted graphs, another array that stores the weights of all edges is accordingly required.

We apply both coarse-grained (that one block processes one root vertex s) and fine-grained parallel (that threads within the block compute shortest paths and dependencies that related to s) strategies. The pseudo-code in this paper describes the parallel procedure of threads within a block. Algorithm 1 shows the initialization of required variables. U and F represent unsettled set and frontier set, respectively. v is unsettled if $U[v] = 1$ and is frontier node if $F[v] = 1$. d

represents the tentative distance and $\sigma[v]$ is the number of shortest paths from s to v . $\delta[v]$ stores the dependencies of v . S and $ends$ record the levels of traversal as CSR format and they are used in the dependency accumulation step. As can be seen in Algorithm 3, in the dependency accumulation part, we get nodes at the same level from S and $ends$ and accumulate dependencies of these nodes simultaneously. Note that in Algorithm 3 we only assign threads for nodes that need to be inspected rather than assign for all nodes, which enhances the efficiency by avoiding redundant threads.

Algorithm 1 Betweenness Centrality: Variable Initialization

```

1: for  $v \in V$  do in parallel
2:    $U[v] \leftarrow 1$ 
3:    $F[v] \leftarrow 0$ 
4:    $d[v] \leftarrow \infty$ 
5:    $\sigma[v] \leftarrow 0$ 
6:    $\delta[v] \leftarrow 0$ 
7:    $ends[v] \leftarrow 0$ 
8:    $S[v] \leftarrow 0$ 
9: end for
10:  $d[s] \leftarrow 0$ 
11:  $\sigma[s] \leftarrow 1$ 
12:  $U[s] \leftarrow 0$ 
13:  $F[s] \leftarrow 1$ 
14:  $S[0] \leftarrow s; S_{len} \leftarrow 1$ 
15:  $ends[0] \leftarrow 0; ends[1] \leftarrow 1; ends_{len} \leftarrow 2$ 
16:  $\Delta \leftarrow 0$ 

```

Algorithm 2 Betweenness Centrality: Shortest Path Calculation by Dijkstra Algorithm

```

1: while  $\Delta < \infty$  do
2:   for  $v \in V$  and  $F[v] = 1$  do in parallel
3:     for  $w \in neighbors(v)$  do
4:       if  $U[w] = 1$  and  $d[v] + weight_{vw} < d[w]$  then
5:          $d[w] \leftarrow d[v] + weight_{vw}$ 
6:          $\sigma[w] \leftarrow 0$ 
7:       end if
8:       if  $d[w] = d[v] + weight_{vw}$  then
9:          $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$ 
10:      end if
11:    end for
12:  end for
13:   $\Delta \leftarrow \infty$ 
14:  for  $v \in V$  do in parallel
15:    if  $U[v] = 1$  and  $d[v] < \infty$  then
16:       $atomicMin(\Delta, d[v] + \Delta_v)$ 
17:    end if
18:  end for
19:   $cnt \leftarrow 0$ 
20:  for  $v \in V$  do in parallel
21:     $F[v] \leftarrow 0$ 
22:    if  $U[v] = 1$  and  $d[v] < \Delta$  then
23:       $U[v] \leftarrow 0$ 
24:       $F[v] \leftarrow 1$ 
25:       $t \leftarrow atomicAdd(S_{len}, 1)$ 
26:       $S[t] \leftarrow v$ 
27:       $atomicAdd(cnt, 1)$ 
28:    end if
29:  end for
30:  if  $cnt > 0$  then
31:     $ends[ends_{len}] \leftarrow ends[ends_{len} - 1] + cnt$ 
32:     $ends_{len} \leftarrow ends_{len} + 1$ 
33:  end if
34: end while

```

Algorithm 3 Betweenness Centrality: Dependency Accumulation

```

1:  $depth \leftarrow ends_{len} - 1$ 
2: while  $depth > 0$  do
3:    $start \leftarrow ends[depth - 1]$ 
4:    $end \leftarrow ends[depth] - 1$ 
5:   for  $0 \leq i \leq end - start$  do in parallel
6:      $w \leftarrow S[start + i]$ 
7:      $dsw \leftarrow 0$ 
8:     for  $v \in neighbors(w)$  do
9:       if  $d[v] = d[w] + weight_{wv}$  then
10:         $dsw \leftarrow dsw + \sigma[w]/\sigma[v] * (1 + \delta[v])$ 
11:      end if
12:    end for
13:     $\delta[w] \leftarrow dsw$ 
14:    if  $w \neq s$  then
15:       $atomicAdd(CB[w], \delta[w])$ 
16:    end if
17:  end for
18:   $depth \leftarrow depth - 1$ 
19: end while

```

3.2 Parallel Dijkstra algorithm

The parallel version of DFS procedure, which is used in BC algorithm for unweighted network, could be modified naturally from its sequential version because vertices located at the same level of the DFS tree can be inspected simultaneously. While for Dijkstra algorithm, picking one frontier node each time makes its parallelization a difficult task. However, this restriction can be relaxed, which means that several nodes could be settled becoming frontier set and be inspected simultaneously in the next step. In this paper, we apply the method described in [23, 16]. In this method, $\Delta_{node\ v} = \min(w(v, u) : (v, u) \in E)$ is precomputed. Then we define Δ_i as

$$\Delta_i = \min\{(D(u) + \Delta_{node\ u}) : u \in U_i\}, \quad (2)$$

where $D(u)$ is the tentative distance of node u , U_i is the unsettled nodes set in iteration i . All nodes that satisfy the following condition

$$D(v) \leq \Delta_i \quad (3)$$

are settled and become frontier nodes. When applying Dijkstra algorithm in betweenness centrality calculation, the number of shortest paths should be counted. To achieve this goal, the above condition should be modified to

$$D(v) < \Delta_i. \quad (4)$$

Fig. 1(a) demonstrates an example, in which vertex v_0 is the source node. If applying Eq. 3, v_1 and v_2 will be frontier nodes after inspecting v_0 in the first iteration, and the number of shortest paths will be 1 for both v_1 and v_2 . Then v_1 and v_2 will be inspected simultaneously in next step. If processing v_2 first, the number of shortest paths for v_3 will be set to 1, while the the correct value of shortest paths' number for v_3 should be 2. This mistake comes from the overambitious condition and v_2 should not be settled after the first iteration. Although the distance will be correct for all nodes using Eq. 3, but the number of shortest paths will be wrong. However, Eq. 4 will lead to correct shortest paths number for v_3 by only settling v_1 after first iteration. This condition could be found at Line 22 in Algorithm 2.

Algorithm 2 depicts our parallel Dijkstra algorithm in detail. The tentative distance and number of shortest paths are calculated which can be seen from Line 2 to Line 12. In this part, there will be a subtle parallel problem that several nodes in the frontier set may connect to the same node, as can be seen in Fig. 1(b). In this example, both v_1 and v_2 are in frontier set and connect to w , which results in the classical race condition problem. To avoid this situation, we define a lock for each node. The first thread focus on w will achieve the lock and other threads will not be permitted to change $d[w]$ and $\sigma[w]$. Note that other threads must not wait because in CUDA framework, a group of threads in a warp performs as a SIMD (Single Instruction Multiple Data) unit. After computing d and σ for all nodes, we can achieve Δ_i based on the above analysis, as can be seen from Line 13 to Line 18. In the end, U , F , S and $ends$ are updated for next iteration.

3.3 Work-efficient method

As can be seen on Line 2 in Algorithm 2, threads will be assigned to all nodes but only nodes that in the frontier set will perform the calculation job, which may be inefficient. McLaughlin et al. figured out an excellent work-efficient technique to solve this problem [14]. Here we develop

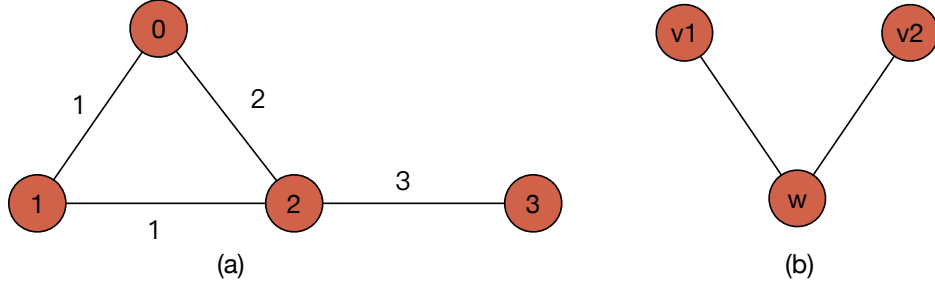


Figure 1. (a) An example of choosing frontier nodes, in which using Eq. 3 will make the number of shortest paths of v_3 incorrect. (b) An example of race condition. v_1 and v_2 are frontier nodes in one iteration, and both of which are connected with w .

our work-efficient version by employing this technique. F will be changed to a *queue* that stores all frontier nodes and a variable F_{len} is defined to recode the length of F , as can be seen in Algorithm 4. Then on Line 2 in Algorithm 5, threads can be assigned to $F[0] \sim F[F_{len} - 1]$, which may be much smaller than the total number of nodes. At the same time, the method of updating F should also be changed, which can be seen in Algorithm 5.

Algorithm 4 Work-efficient betweenness Centrality: Variable Initialization

```

1: for  $v \in V$  do in parallel
2:   // initialize other variables except  $F$ 
3: end for
4:  $F[0] \leftarrow s$ 
5:  $F_{len} = 1$ 
6: // initialize other variables

```

Algorithm 5 Work-efficient betweenness Centrality: Shortest Path Calculation by Dijkstra Algorithm

```

1: while  $\Delta < \infty$  do
2:   for  $0 \leq i < F_{len}$  do in parallel
3:      $v \leftarrow F[i]$ 
4:     // inspect  $v$ 
5:   end for
6:   // calculate  $\Delta$ 
7:    $F_{len} \leftarrow 0$ 
8:   for  $v \in V$  do in parallel
9:     if  $U[v] = 1$  and  $d[v] < \Delta$  then
10:       $U[v] \leftarrow 0$ 
11:       $t \leftarrow \text{atomicAdd}(F_{len}, 1)$ 
12:       $F[t] \leftarrow v$ 
13:    end if
14:  end for
15:  if  $F_{len} > 0$  then
16:     $ends[ends_{len}] \leftarrow ends[ends_{len} - 1] + F_{len}$ 
17:     $ends_{len} \leftarrow ends_{len} + 1$ 
18:    for  $0 \leq i < F_{len}$  do
19:       $S[S_{len} + i] \leftarrow F[i]$ 
20:    end for
21:     $S_{len} \leftarrow S_{len} + F_{len}$ 
22:  end if
23: end while

```

3.4 Warp-centric method

Real world networks always have scale-free character, which means their degree distributions follow power law. When implementing parallel graph algorithms through node parallel strategy, this feature brings in serious load-imbalance problem. Most nodes have low degrees while some nodes have extremely high degrees. Threads that assigned to high degree nodes will run slowly and other threads have to wait. Edge parallel strategy can solve this problem [24] but bring in other under-utilizations at the same time. In this paper, we apply the novel warp-centric

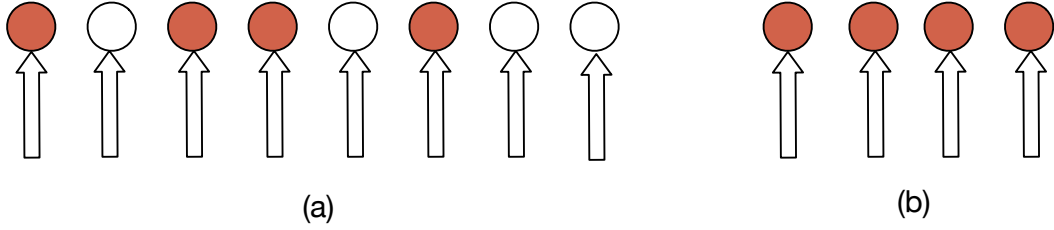


Figure 2. An example of threads allocation in node parallel method (a) and work-efficient method (b). Red nodes are frontier nodes that should be processed and an arrow represents a warp that be assigned to the corresponding node. Warp-centric method will waste more threads on nodes that do not need to be inspected. But combining warp-centric and work-efficient method can solve this problem, as shown in (b).

method [19], which allocates a warp to one node rather than a thread. Then threads within a warp focus on part of edges connected the specific node. As a result, each thread does less job for high degree nodes and the waiting time will be sharply decreased. Moreover, memory access patterns can be more coalesced than the conventional thread-level task allocation and because of this, the efficiency of memory access can be essentially improved.

Nevertheless, the warp-centric method also has some drawbacks. Firstly, node degree may be smaller than the warp size, which is always 32 in modern GPU. To solve this problem, virtual warps are proposed in [19]. Secondly, the number of required threads will be raised as each node needs *WARP_SIZE* threads rather than one thread in this situation. But the number of threads in one block is fixed, hence each thread will be assigned to more nodes iteratively, which may results in low performance. We find that work-efficient method can relieve this problem because it requires less threads compared to the conventional node-parallel method, as can be seen in Fig. 2. In this paper, we apply the warp-centric method for both node-parallel and work-efficient method. As a result, we get four algorithms (see Fig. 3) that using different threads allocation strategies and we compare them on both real-world and synthetic networks.

4 Experiments

4.1 Networks and settings

We collect six real-world networks from the Internet, which have broad types including collaboration network, epinions trust network, email communication network, wiki vote network and

two biological networks. They are publicly available in the Internet and have been analyzed extensively by previous literatures [25, 26, 27, 28, 29]. The details of these networks are listed in Table 1. To further understand the effect of network structures to algorithms’ performance, we generate two types of networks, which are Erdős–Rényi (ER) random graphs [30] and Kronecker graphs [31]. The degree distribution of ER random graph is Poisson, indicating its nodes’ degrees are relatively balanced. While Kronecker graph possesses scale-free and small-world characters, which make it more similar to the realistic network. The two biological networks and the cond-mat-2005 collaboration network are weighted networks and for other networks, we uniformly assign random edge weights ranging from 1 to 10. We run the four GPU methods on Geforce GTX 980 (only entry-level for scientific computing) using CUDA 7.5 Toolkit. We also develop the sequential algorithm using C++ and optimize it by applying binary heap in Dijkstra algorithm due to Fibonacci heap’s inefficiency in practical use, making our CPU version BC algorithm performs better than most of the existing implementations. And we run the CPU version algorithm on competent Intel Xeon E5620 with 2.40GHz.

Table 1. Details of networks from public dataset

Network	Vertices	Edges	Max degree	Average degree	Description
bio-human-gene1 [32, 28]	22283	12345963	7940	1108.11	Human gene regulatory network
bio-human-gene2 [32, 28]	14340	9041364	7230	1261.00	Human gene regulatory network
cond-mat-2005 [29]	39577	175693	278	8.88	Collaboration network
email enron [33, 26]	36692	183831	1383	10.02	Email communication network from Enron
soc epinions1 [33, 25]	75879	405740	3044	10.69	Who-trusts-whom network of Epinions.com
wiki vote [33, 27]	7115	100762	1065	28.32	Wikipedia who-votes-on-whom network

4.2 Results

From Fig. 3, we can see that all the four GPU programs achieve much better performance than the CPU version on all the six real-world networks. The algorithm that applies work-efficient coupled with warp-centric technique is the best one for achieving $30\times$ to $150\times$ speedup and its performance could be essentially improved on more sophisticated GPU devices. Work-efficient method is more efficient than node-parallel in all networks, while warp-centric method is better on large degree networks, such as the two biological networks. For networks with low average degrees, applying warp-centric method alone is always inefficient because nodes’ degrees are always smaller than *WARP_SIZE*. Using smaller virtual *WARP_SIZE* could be better on these networks and we will demonstrate this hypothesis later. However, combining

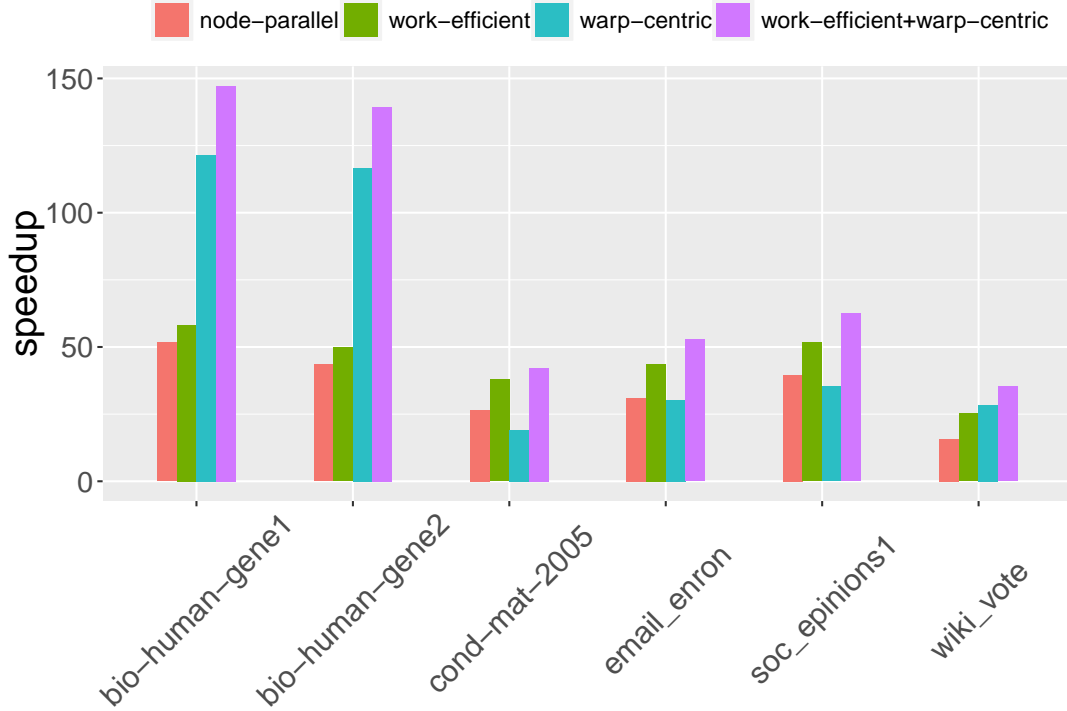


Figure 3. Speedups over sequential algorithm of the four GPU implementations on realistic networks. We define speedup as the quotient of the CPU and GPU algorithm running time. The *WARP_SIZE* is fixed to 32 in the two warp-centric methods.

warp-centric method and work-efficient method is always better than using work-efficient method alone because it needs less threads in each step, which accordingly relieves the influence of the second drawback of warp-centric method. The combination algorithm achieves $2.65\times$ speedup over the baseline node-parallel strategy on average.

To deeply mining the relationship of the network structure and the performance of the four GPU implementations, we further run them on two types of synthetic graphs, as can be seen in Fig. 4. From Fig. 4(a), (b), (c) and (d), we find that work-efficient algorithm works better than node-parallel algorithm in all networks since it always reduces the required number of threads. As can be seen in Fig. 4(a) and (b), warp-centric method works well on networks with large degrees, which is consistent with the conclusion in realistic networks. Note that for Kronecker graphs, warp-centric method works better than that for random graphs as Kronecker graphs have serious load-imbalance problem and warp-centric technique appropriately solves it. While for ER random graphs in 4(a), the advantage of warp-centric method is only the efficient memory

access. For low degree graphs, warp-centric method works even worse than node-parallel strategy because the degrees are always smaller than *WARP_SIZE*, as can be seen in Fig. 4(c) and (d). For random graphs, the performance of warp-centric method is extremely poor when the average degree is smaller than 8 and Fig. 4(e) explains the reason. The small average degree brings in large average depth, which means that the average size of the frontier sets is small. In this case, warp-centric method assigns more useless threads to nodes that do not need inspections. On the contrary, as the degree grows, it is closer to *WARP_SIZE* and the depths drop down sharply, which make the warp-centric method performs much better. While low-degree Kronecker graphs have power-law degree distributions and small average depths, which make warp-centric method works not as bad as on random graphs. However, the combination of the two methods always runs faster than applying work-efficient method alone because it avoids the second drawback of warp-centric method, which is discussed in the previous section. In conclusion, work-efficient method always achieves better performance while the performance of warp-centric method relies on networks' structures but the joint version always achieves the best performance.

From the above analysis, applying smaller *WARP_SIZE* may accelerate the two implementations which using warp-centric method when the networks' average degree is small. And this hypothesis can be verified in Fig. 5. We apply smaller *WARP_SIZE* on email enron network, soc.epinions1 network and other two synthetic graphs whose average degrees are both four. From Fig. 5(a) and (b), we find that implementations with smaller *WARP_SIZE* do perform better than both of the baseline node-parallel algorithms and the large *WARP_SIZE* algorithm on both of the low-degree realistic networks. And when coupled with work-efficient method, algorithms with smaller *WARP_SIZE* also perform better than both of the work-efficient strategy alone and the combination of work-efficient and large *WARP_SIZE*. The reason is that small *WARP_SIZE* reduces the required number of threads and then eliminates the waste of assigning more threads to a node than its degree. The implementations which have small *WARP_SIZE* and coupled with work-efficient method achieve the best performance because they avoid both drawbacks of warp-centric method but utilize the advantages of this technique. The results on low-degree Kronecker graph is similar as on realistic networks since Kronecker graph is similar with real-world network. For ER random graphs, the algorithm with small *WARP_SIZE* does not achieve better performance compared to node-parallel version because the large average depth, which is analyzed in previous section. However, when coupled with work-efficient method, the implementations with small *WARP_SIZE* perform slightly better than the work-efficient algorithm, which further proves the excellence and stability of the joint algorithm. In summary, the joint algorithm are most efficient and insensitive to network structure. And if we

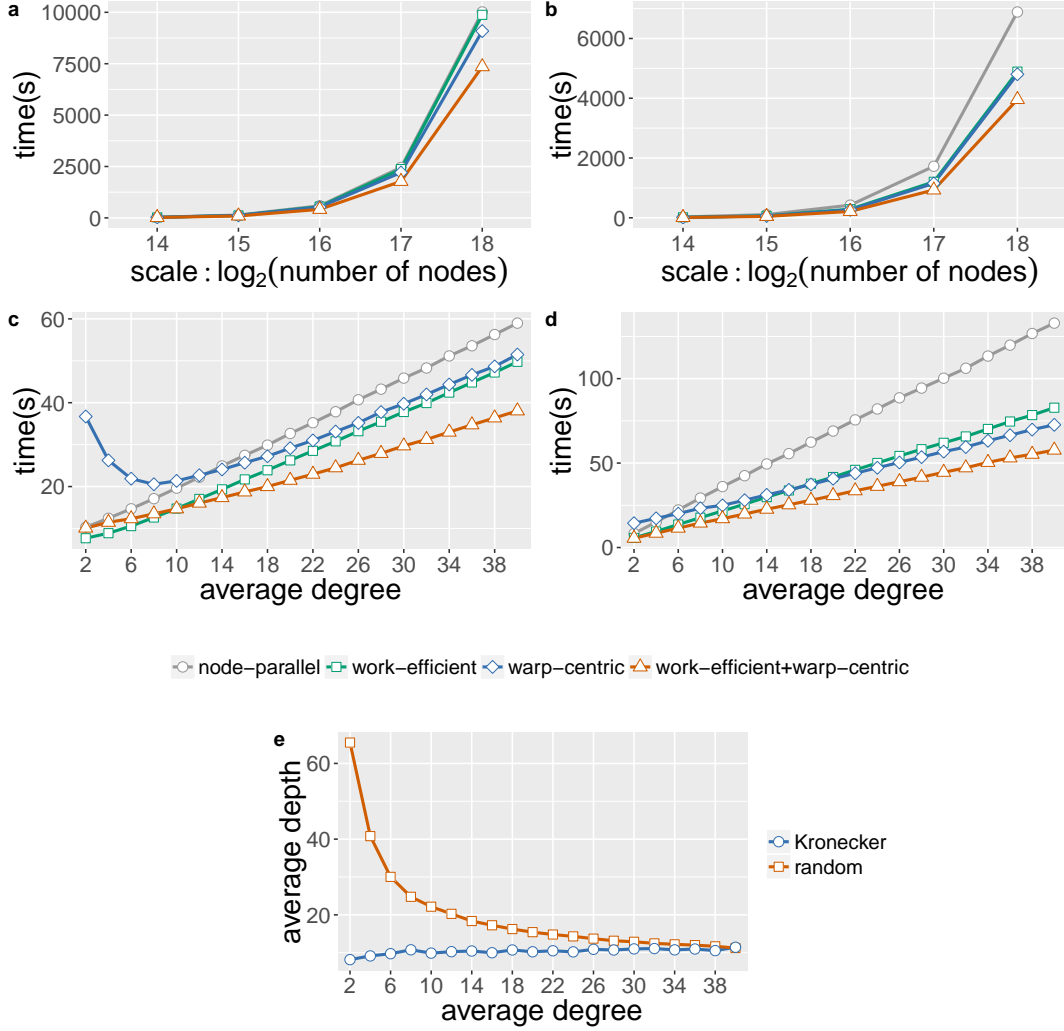


Figure 4. Performance of the four implementations on ER random and Kronecker graphs. The *WARP_SIZE* is fixed to 32 in the two warp-centric methods. (a) and (b) tune the number of nodes from 2^{14} to 2^{18} for ER random and Kronecker graphs, respectively. And the average degrees are fixed to 32 for both of the two types of networks. (c) and (d) separately tune the average degrees for random and Kronecker networks, in which the random networks have 20000 vertices and the Kronecker networks have 2^{15} nodes. (e) illustrates the average depths of search trees for random graphs used in (c) and Kronecker graphs used in (d). Networks with larger depths have smaller average frontier sets, indicating the poor performance with parallelism.

choose an appropriate *WARP_SIZE*, its performance could be even better.

5 Conclusion

Existing GPU version of betweenness centrality algorithms only concentrate on unweighted networks for simplification. Our work that computing betweenness centrality on large weighted networks bridges this gap and achieves prominent efficiency enhancement compared to the CPU implementation. Moreover, we apply two excellent techniques which are work-efficient and warp-centric methods in our algorithm. Work-efficient method allocates threads more efficiently and warp-centric method solves the load imbalance problem and simultaneously optimizes the memory access. We compare these implementations with CPU algorithm and the basic GPU algorithm in realistic networks. The results show that GPU parallel algorithms perform much better than the sequential algorithm and the algorithm which integrates the two techniques is the best, achieving $30\times$ to $150\times$ speedup over the CPU version. Results on generated random graphs and Kronecher graphs further justify the outperformance of our solution.

For future work, we will consider implementing GPU algorithm to process dynamic networks. When networks changes a little (like few new nodes come in or several links vanish), calculating betweenness centrality for all nodes is unnecessary because betweenness centrality of most nodes and edges will not be changed. Some previous works have explored the sequential algorithm on this issue [34, 35, 36]. We plan to develop GPU version of these algorithms to achieve better performance.

Acknowledgments

This work was supported by NSFC (Grant No. 71501005) and the fund of the State Key Lab of Software Development Environment (Grant Nos. SKLSDE-2015ZX-05 and SKLSDE-2015ZX-28). R. F. also thanks the Innovation Foundation of BUAA for PhD Graduates.

References

- [1] FREEMAN LC. A Set of Measures of Centrality based on betweenness. *Sociometry*. 1977;40(1):35–41.
- [2] Barthélemy M. Betweenness centrality in large complex networks. *The European Physical Journal B-Condensed Matter and Complex Systems*. 2004;38(2):163–168.

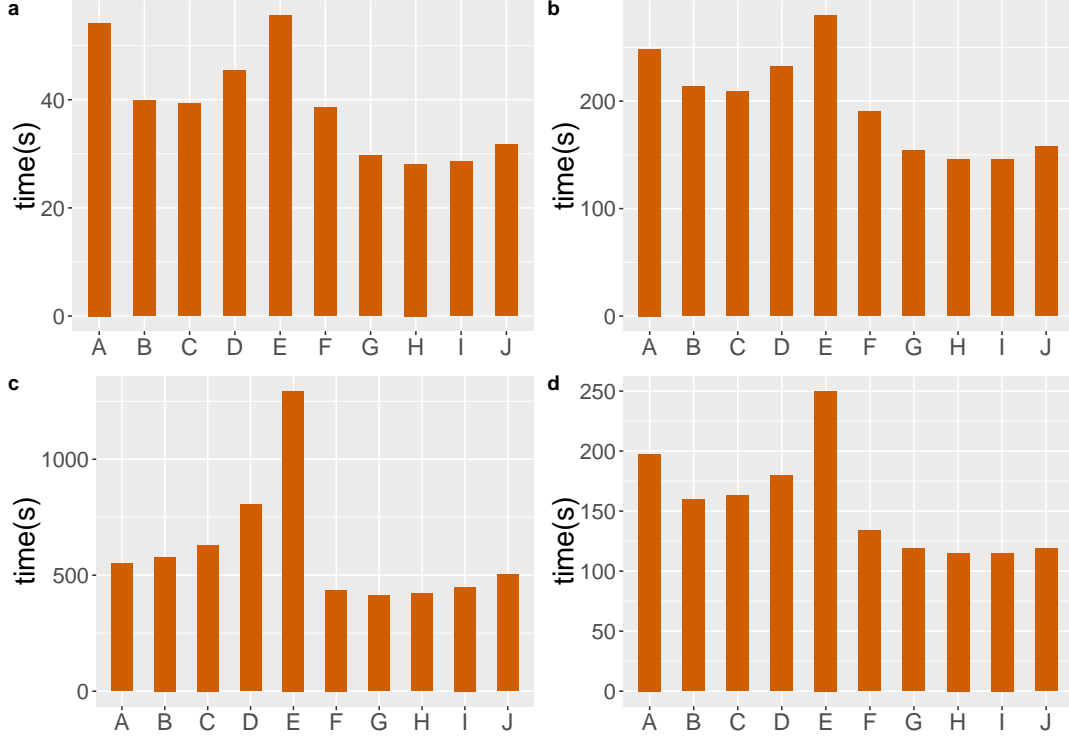


Figure 5. Applying other *WARP_SIZE* on several low-degree networks. A-J represent node-parallel, warp4, warp8, warp16, warp32, work-efficient, work-efficient+warp4, work-efficient+warp8, work-efficient+warp16, work-efficient+warp32, respectively. (a) and (b) are email enron network and soc_epinions1 network, respectively, on which smaller warp size achieves better performance than both node-parallel method and the algorithm with large *WARP_SIZE*. (c) is a random graph with 2^{17} nodes whose average degree is four. Warp-centric method can not accelerate the speed when combining node-parallel strategy. But when combining small *WARP_SIZE* with work-efficient method, the performance will be slightly better than applying work-efficient method alone. (d) is Kronecker graph with 2^{17} nodes and the average degree is four, on which smaller *WARP_SIZE* achieves better performance.

- [3] Goh KI, Oh E, Kahng B, Kim D. Betweenness centrality correlation in social networks. *Phys Rev E*. 2003;67:017101.
- [4] Girvan M, Newman MEJ. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*. 2002;99(12):7821–7826.
- [5] Leydesdorff L. Betweenness centrality as an indicator of the interdisciplinarity of scientific journals. *Journal of the American Society for Information Science and Technology*. 2007;58(9):1303–1319.
- [6] Motter AE, Lai YC. Cascade-based attacks on complex networks. *Phys Rev E*. 2002;66:065102. doi:10.1103/PhysRevE.66.065102.
- [7] Brandes U. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*. 2001;25(2):163–177.
- [8] Merrill D, Garland M, Grimshaw A. High-Performance and Scalable GPU Graph Traversal. *ACM Trans Parallel Comput*. 2015;1(2).
- [9] Wang Y, Davidson A, Pan Y, Wu Y, Riffel A, Owens JD. Gunrock: A High-performance Graph Processing Library on the GPU. In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP 2015. New York, NY, USA: ACM; 2015. p. 265–266.
- [10] Harish P, Narayanan PJ. Accelerating Large Graph Algorithms on the GPU Using CUDA. In: *Proceedings of the 14th International Conference on High Performance Computing*. HiPC’07. Berlin, Heidelberg: Springer-Verlag; 2007. p. 197–208.
- [11] Cong G, Bader DA. An Experimental Study of Parallel Biconnected Components Algorithms on Symmetric Multiprocessors (SMPs). In: *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. IPDPS ’05. Washington, DC, USA: IEEE Computer Society; 2005. p. 45b–45b.
- [12] Shi Z, Zhang B. Fast network centrality analysis using GPUs. *BMC Bioinformatics*. 2011;12:149.
- [13] Sariyüce AE, Kaya K, Saule E, Çatalyürek ÜV. Betweenness Centrality on GPUs and Heterogeneous Architectures. In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*; 2013. p. 76–85.

- [14] McLaughlin A, Bader DA. Scalable and high performance betweenness centrality on the GPU. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2014. p. 572–583.
- [15] Martín PJ, Torres R, Gavilanes A. CUDA Solutions for the SSSP Problem. In: Proceedings of the 9th International Conference on Computational Science. Springer-Verlag; 2009. p. 904–913.
- [16] Ortega-Arranz H, Torres Y, Llanos DR, Gonzalez-Escribano A. A New GPU-based Approach to the Shortest Path Problem. In: High Performance Computing and Simulation; 2013. p. 505–511.
- [17] Delling D, Goldberg AV, Nowatzyk A, Werneck RF. PHAST: Hardware-Accelerated Shortest Path Trees. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium. IEEE Computer Society; 2011. p. 921–931.
- [18] Davidson A, Baxter S, Garland M, Owens JD. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium. IEEE Computer Society; 2014. p. 349–359.
- [19] Hong S, Kim SK, Oguntebi T, Olukotun K. Accelerating CUDA Graph Algorithms at Maximum Warp. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming; 2011. p. 267–276.
- [20] Dijkstra EW. A Note on Two Problems in Connexion with Graphs. *Numer Math.* 1959;1(1):269–271.
- [21] Floyd RW. Algorithm 97: Shortest Path. *Commun ACM.* 1962;5(6).
- [22] Bell N, Garland M. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis; 2009. p. 18:1–18:11.
- [23] Crauser A, Mehlhorn K, Meyer U, Sanders P. A Parallelization of Dijkstra’s Shortest Path Algorithm. In: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science; 1998. p. 722–731.
- [24] Jia Y, Lu V, Hoberock J, Garland M, Hart JC. Edge v. node parallelism for graph centrality metrics. *GPU Computing Gems.* 2011;2:15–30.

- [25] Richardson M, Agrawal R, Domingos P. Trust Management for the Semantic Web. In: Proceedings of the Second International Conference on Semantic Web Conference. LNCS-ISWC'03; 2003. p. 351–368.
- [26] Leskovec J, Lang KJ, Dasgupta A, Mahoney MW. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*. 2009;6(1):29–123.
- [27] Leskovec J, Huttenlocher D, Kleinberg J. Signed Networks in Social Media. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '10; 2010. p. 1361–1370.
- [28] Bansal M, Belcastro V, Ambesi-Impiombato A, Di Bernardo D. How to infer gene networks from expression profiles. *Molecular systems biology*. 2007;3(1).
- [29] Newman ME. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*. 2001;98(2):404–409.
- [30] Erdős P, A R. On random graphs. *Publicationes Mathematicae*. 1959;6:290–297.
- [31] Leskovec J, Chakrabarti D, Kleinberg J, Faloutsos C, Ghahramani Z. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*. 2010;11:985–1042.
- [32] Rossi RA, Ahmed NK. The Network Data Repository with Interactive Graph Analytics and Visualization. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence; 2015. Available from: <http://networkrepository.com>.
- [33] Leskovec J, Krevl A. SNAP Datasets: Stanford Large Network Dataset Collection; 2014. <http://snap.stanford.edu/data>.
- [34] Lee MJ, Choi S, Chung CW. Efficient Algorithms for Updating Betweenness Centrality in Fully Dynamic Graphs. *Inf Sci*. 2016;326(C):278–296.
- [35] Singh RR, Goel K, Iyengar S, Gupta S. A faster algorithm to update betweenness centrality after node alteration. *Internet Mathematics*. 2015;11(4-5):403–420.
- [36] Nasre M, Pontecorvi M, Ramachandran V. Betweenness centrality—incremental and faster. In: International Symposium on Mathematical Foundations of Computer Science. Springer; 2014. p. 577–588.